USENIX Security 21

ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems

论文分析 by 蒋隽

2022.07.24

Contents

Background

- 漏洞行为比漏洞形成的根本原因更容易被检测到

- 开发人员难以定位检测到安全风险的漏洞位置

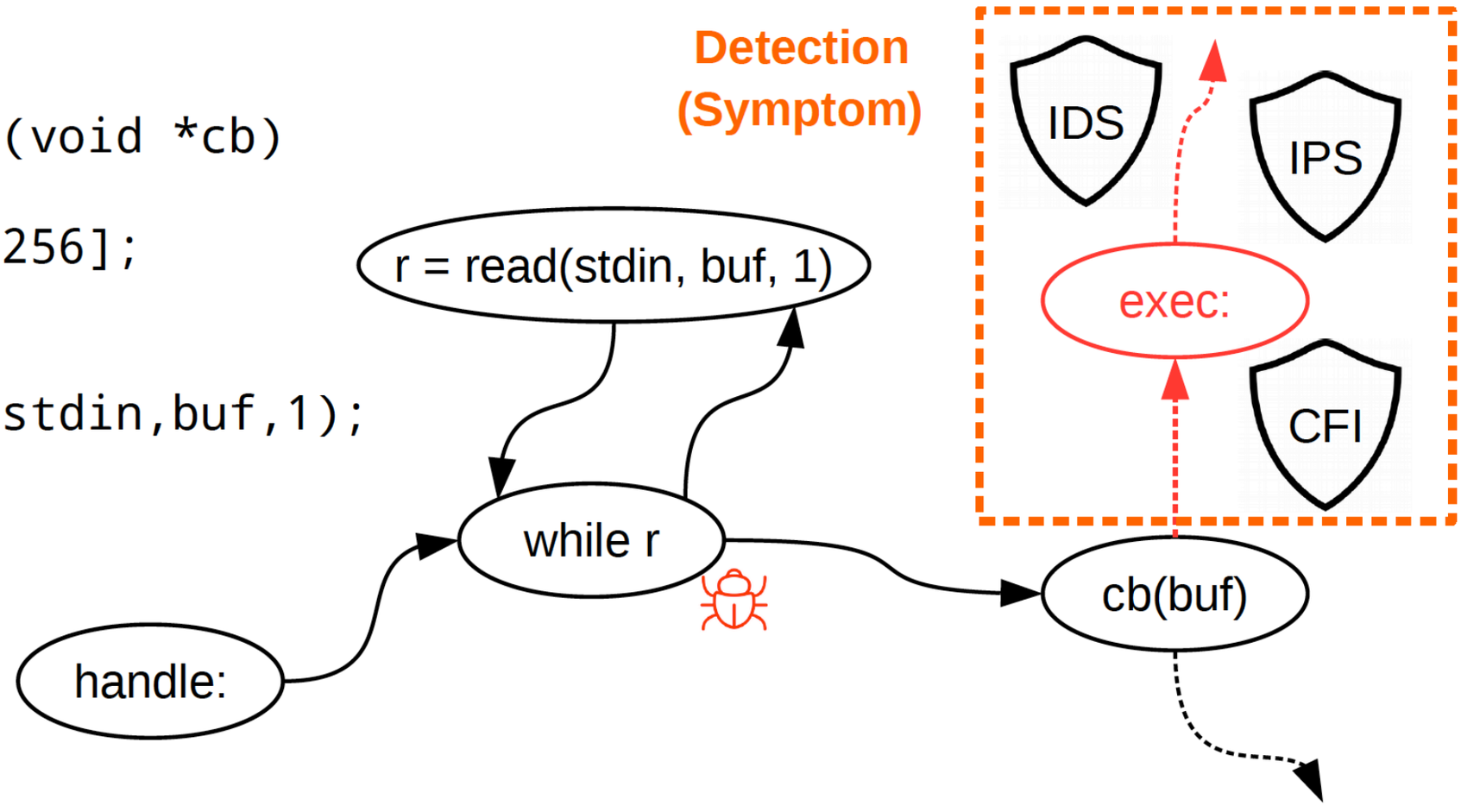
- 发布补丁仅考虑表面修补而不是根本症结

ARCUS —— an automated framework for localizing the root cause of vulnerabilities in executions flagged by end-host runtime monitors.

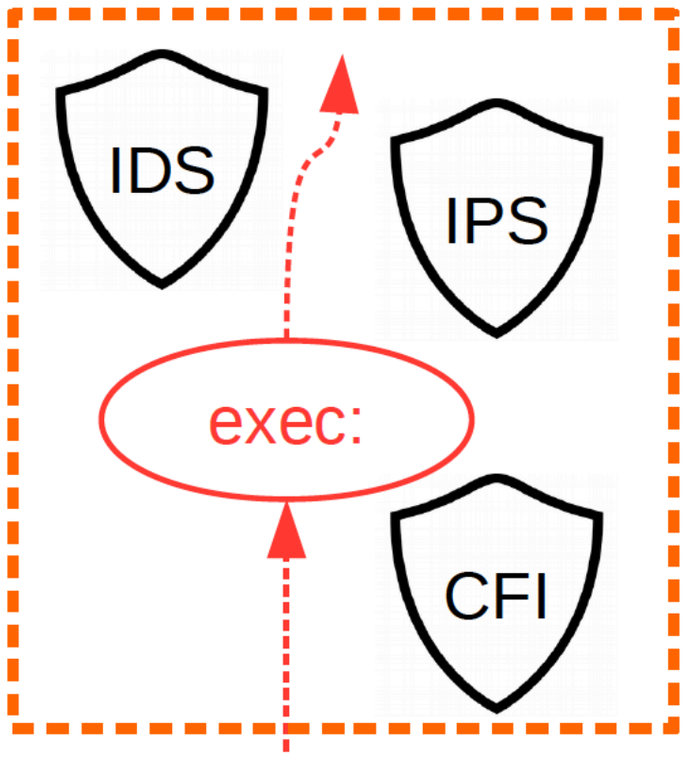
A Toy Case

```
void handle(void *cb)
{
  char buf[256];
  int r=1;
  while r
    r=read(stdin,buf,1);
  cb(buf);
}
```


User Program



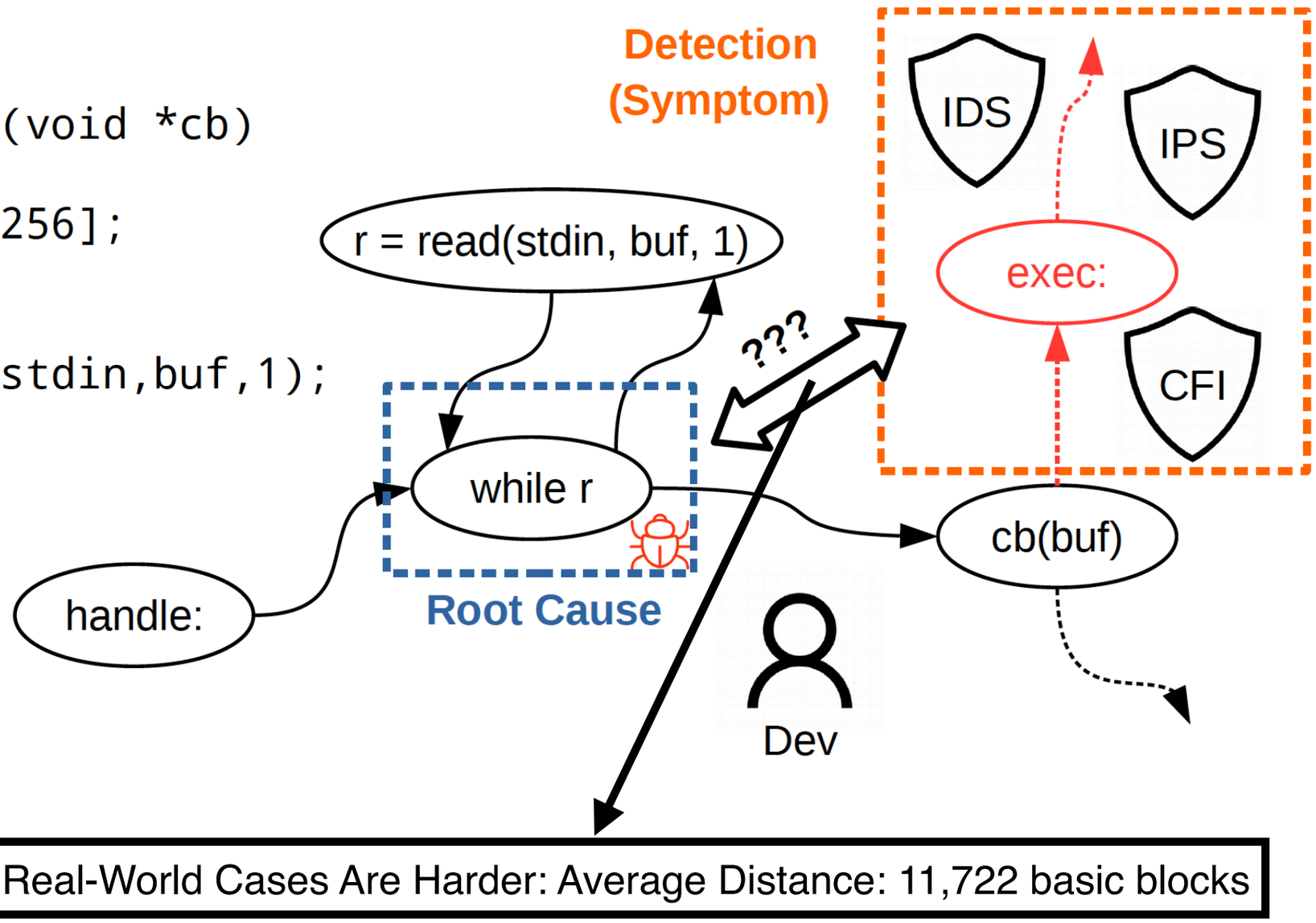
Why Symptoms?



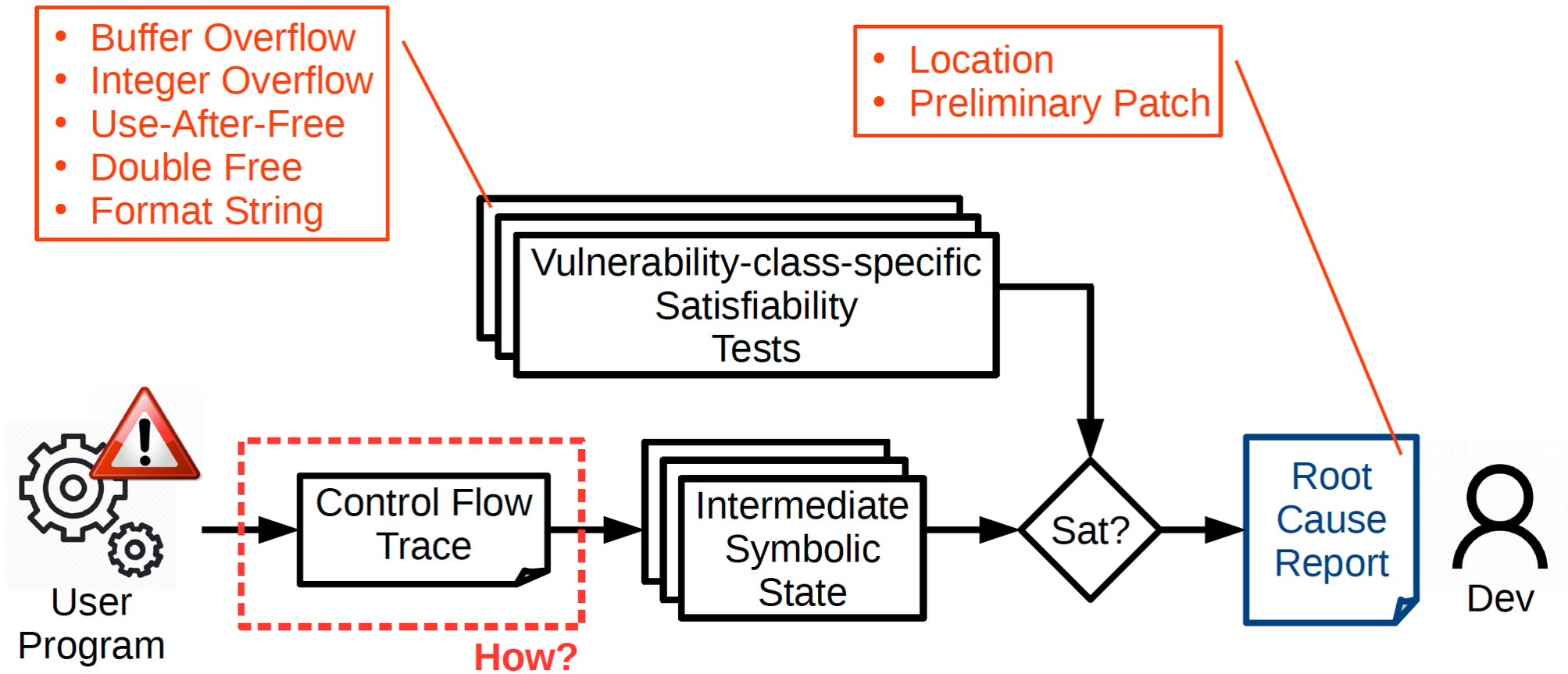
- Easiest to detect
  - Manifestation of behavior
- But how do we fix it?
  - Input filters
  - Function hardening
- Brittle, expensive

Ideal Fix: Developer Patch

```
void handle(void *cb)
{
  char buf[256];
  int r=1;
  while r
    r=read(stdin,buf,1);
  cb(buf);
}
```

ARCUS Pipeline



Example: CVE-2018-12327

```
$ ./ntpq -4 [`python -c 'print "A" * 300'`]
Name or service not known
*** stack smashing detected ***: <unknown> terminated
```

The State Explosion Problem

```
1.  int openhost(const char *hname, ...) {
2.    char *cp;
3.    char name[256];
4.
5.    cp = hname;
6.    if (*cp == '[') {
7.      cp++;
8.      for (i = 0; *cp && *cp != ']'; cp++, i++)
9.        name[i] = *cp;
10.     if (*cp == ']') {
11.       name[i] = '\0';
12.       hname = name;
13.     } else return 0;
14.     /* [...] */
```

Symbols

```
hname    := [s1,s2,…]
name     := []
cp       := {}
ret_ptr  := {c1}
```

How many times should Line 9 iterate?

Solution: Control Flow Trace

```
1. int openhost(const char *hname, ...) {
2.  char *cp;
3.  char name[256];
4.
5.  cp = hname;
6.  if (*cp == '[') {
7.   cp++;
8.   for (i = 0; *cp && *cp != ']'; cp++, i++)
9.    name[i] = *cp;
10.  if (*cp == ']') {
11.   name[i] = '\0';
12.   hname = name;
13.  } else return 0;
14.  /* [...] */
```

Trace

Taken

Taken

x312

Symbols

```
hname   := ['[',s2,…,']']
name    := [s2,s3,…]
cp      := hname+312
ret_ptr := {s258}
```

Corrupted return pointer!

Localizing Root Cause

Intermediate Symbolic States

```
hname    := ['[',s2,…]
name     := [s2]
cp       := hname+1
ret_ptr  := {c1}
```

.
.
.

```
hname    := ['[',s2,…,']']
name     := [s2,s3,…]
cp       := hname+312
ret_ptr  := {s258}
```

What if we *didn't* corrupt the pointer?

```
hname    := ['[',s2,…]
name     := [s2,s3,…]
cp       := hname+257
ret_ptr  := {c1}
```

exit loop

```
hname    := ['[',s2,…,']']
name     := [s2,s3,…]
cp       := hname+257
ret_ptr  := {c1}
```

What's Different?

```
hname    := ['[',s2,…,']']
name     := [s2,s3,…]
cp       := hname+312
ret_ptr  := {s258}
```

Δ

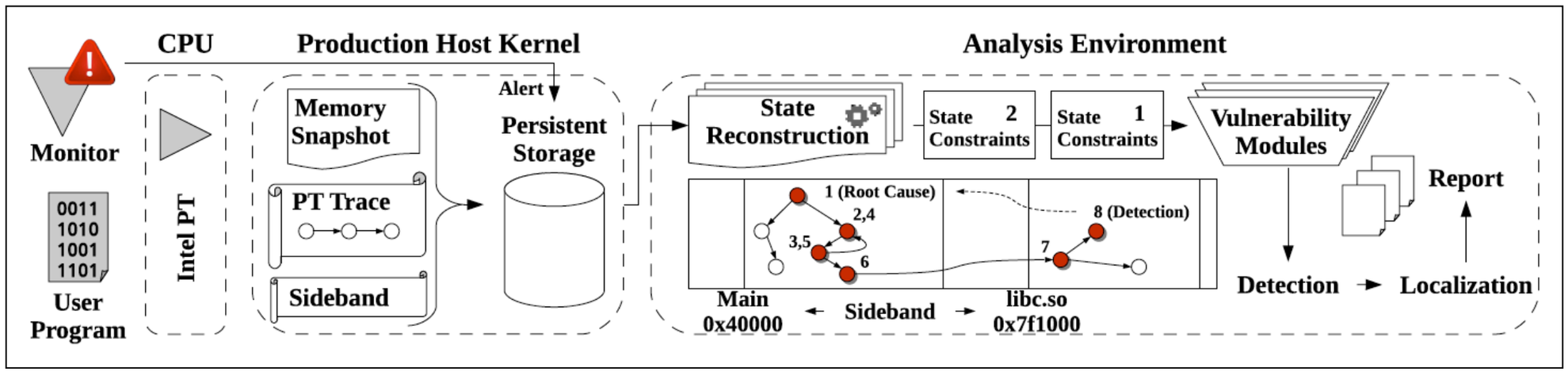
```
hname    := ['[',s2,…,']']
name     := [s2,s3,…]
cp       := hname+257
ret_ptr  := {c1}
```

hname[257] != ‘]’ ⟷ (contradiction) ⟷ hname[257] == ‘]’

Preliminary Patch:

```
for (i=0; *cp && *cp != ']'
&& index(‘]’, hname) <= 257; cp++,i++)
```

ARCUS architecture



- 用户程序在终端主机中执行，而 ARCUS 内核模块使用 Intel PT 对其进行快照和跟踪。

- 当运行时监视器标记违规或异常时，数据将被发送到分析环境，在该环境中重建符号状态，模块在该环境中检测、定位和报告漏洞。

Symbolic Execution Along Traced Paths

- 一旦监视器发出警报，ARCUS 将构建来自内核模块发送的数据的符号程序状态
- 使用符号分析，由于分析探索了不同的路径在程序中，它对符号数值施加约束，监测他们的数值
- 通过符号分析跟踪可以达到程序状态的可能数据值。
- 使用符号分析不是为了静态探索所有可能的路径，而是考虑所有可能的数据都流过一个特定的路径
- 符号化所有可以被攻击者控制的输入数据（命令行参数、环境变量、文件、套接字和其他标准 I/O），只为得到被追踪的路径，这避免了状态爆炸

Analysis Modules——Stack & Heap Overflow

What if this path were to be taken by the program?

- 识别变成符号的代码指针
- 确定编写它的基本块
- 找到控制写块执行的基本块
- 测试这些块上的附加约束是否可能使程序偏离错误行为

Analysis Modules——Stack & Heap Overflow

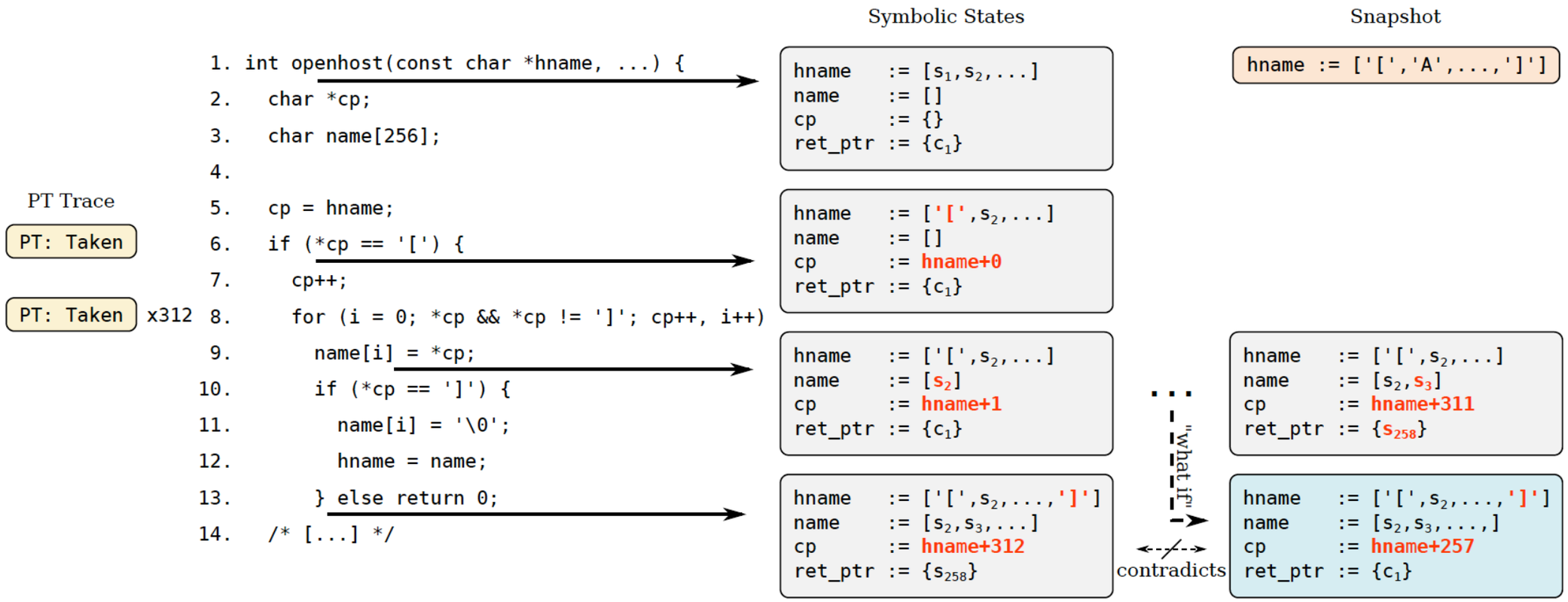
Input: VEX IR statements S starting from last executed.
Tmp n to taint initially.
Result: Addresses A and registers R used to calculate n.

$A \leftarrow \emptyset$
$R \leftarrow \emptyset$
$T \leftarrow \{n\}$
foreach *s in S* **do**
  if Type(*s*) *is Put and* Type(*s.data*) *is RdTmp* **then**
    if $s.data.tmp \in T$ **then**
      $R \leftarrow R \cup \{s.register\}$
    end
  end
  if Type(*s*) *is WrTmp and* $s.tmp \in T$ **then**
    foreach *a in s.data.args* **do**
      if Type(*a*) *is Get* **then**
        $R \leftarrow R \cup \{a.register\}$
      end
      if Type(*a*) *is RdTmp* **then**
        $T \leftarrow T \cup \{a.tmp\}$
      end
      if Type(*a*) *is Load* **then**
        $A \leftarrow A \cup$ EvalTmp(*a.address*)
      end
    end
  end
end

一旦被识别，该模块将通过先前重构的状态向后迭代，以找到包含在识别地址处的数据发生变化的状态。我们将此称为blame 状态。下一步是识别控制它的基本块(guardian)。该模块对重建状态使用前向分析来生成控制依赖图 (CDG) 并找到它们。(如左图)

如果有blame状态的guardian，则根据最短路径选择最接近的状态，并在先状态执行此操作重新访问代码以查看是否存在另一个分支，其约束与blame状态相矛盾。如果发现矛盾的约束，ARCUS 建议在guardian处强制执行。否则，只报告blame状态，因为需要一个全新的guardian。

Revisiting CVE-2018-12327 in more detail.



Analysis Modules——Integer Overflow & Underflow

What if the prior constraint was not satisfiable?

TWO CHALLENGES

- 在没有类型信息的情况下推断寄存器和内存值的符号
- 避免由于开发人员和编译器故意溢出而导致的误报

```
1. TIFFFetchData(TIFF* tif, TIFFDirEntry* dir, char* cp) {
2.   int w = TIFFDataWidth(dir->tdir_type);
3.   tsize_t cc = dir->tdir_count * w;
4.
5.   if (!isMapped(tif)) {
6.     /* [...] */
7.     if (!ReadOK(tif, cp, cc))
8.       goto bad;
9.   /* [...] */
```

```
tdir_count := {s1}
w          := {0,1,2,4,8}
cc         := {}
```

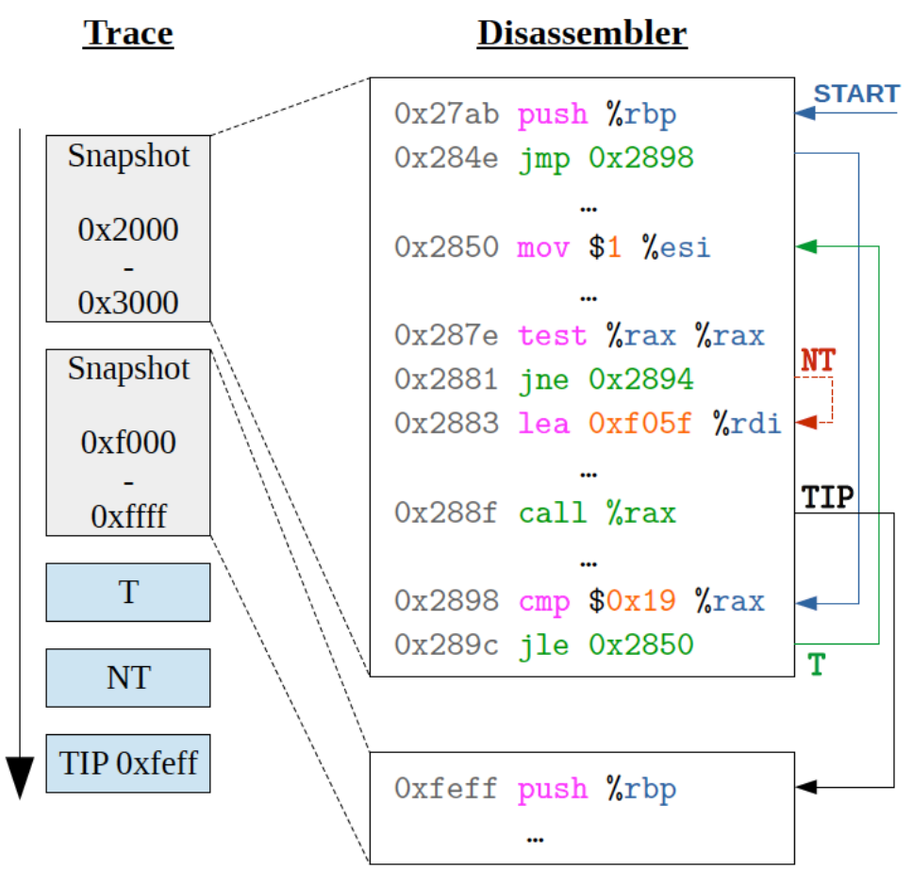
```
tdir_count := {s1}
w          := {0,1,2,4,8}
cc         := {0,...,2^32-1}
cc < w * tdir_count
```

"what if"

```
tdir_count := {0,...,536870911}
w          := {0,1,2,4,8}
cc         := {0,...,2^32-1}
not(cc < w * tdir_count)
```

在这种情况下，攻击者可以制作 TIFF 镜像以溢出保存 cc的寄存器并将其传递给第 7 行的 ReadOK。由于 cc 是两个无符号值的乘积，因此 cc < w * tdir_count 在实际应用中是不可能的，但在第 4 行模块发现它是可满足的，表明 cc 可以溢出。当 cc 然后传递给 ReadOK 时，模块会标记错误。

Capturing the Executed Path



- 采用内核模块管理PT
- 英特尔 PT假设审计员知道已审核的程序，我们的内核模块将其添加到Trace作为快照（灰色数据包）
- 内核模块还捕获和插入在 PT 数据之间动态生成代码页（蓝色地址）
- 对于条件分支，单个采用-未采用位被记录下来。对于间接控制流传输（返回、间接调用和间接跳转）和异步事件（例如，中断、异常），记录目的地
- 英特尔 PT 绕过所有缓存和记忆翻译，最大限度地减少其对跟踪的程序。当为跟踪分配的缓冲区被填满时，CPU 引发一个不可屏蔽中断 (NMI)，内核模块立即处理，因此不会丢失任何数据

Evaluation—— Several Questions

- Is ARCUS accurate at detecting bugs within our covered classes?
- Can ARCUS locate and analyse real-world exploits?
- Are ARCUS' root cause reports consistent with real world advisories and patches?
- Is ARCUS feasible to deploy in terms of runtime and storage overhead?

- Is ARCUS accurate at detecting bugs within our covered classes?

Table 3: RIPE and Juliet Test Cases

Overall Results (Detection by ≥ 1 Strategies)

RIPE	TP	TN	FP	FN	Acc.
BSS	170	170	0	0	100%
Data	190	190	0	0	100%
Heap	190	190	0	0	100%
Stack	260	260	0	0	100%
Juliet	**TP**	**TN**	**FP**	**FN**	**Acc.**
CWE-134	1,200	2,600	0	0	100%
CWE-415	818	2,212	0	0	100%
CWE-416	393	1,222	0	0	100%

By Locating Strategy (RIPE)

Symbolic IP	TP	TN	FP	FN	Acc.
BSS	154	170	0	16	95.3%
Data	171	190	0	19	95.0%
Heap	154	190	0	36	90.5%
Stack	211	260	0	49	90.6%
Int Overflow	**TP**	**TN**	**FP**	**FN**	**Acc.**
BSS	60	170	0	110	67.6%
Data	60	190	0	130	65.8%
Heap	60	190	0	130	65.8%
Stack	150	260	0	110	78.8%

By Locating Strategy (Juliet)

Symbolic Args.	TP	TN	FP	FN	Acc.
CWE-134	1,200	2,600	0	0	100%
Track Frees	**TP**	**TN**	**FP**	**FN**	**Acc.**
CWE-415	818	2,212	0	0	100%
R/W Freed Addrs.	**TP**	**TN**	**FP**	**FN**	**Acc.**
CWE-416	393	1,222	0	0	100%

- ARCUS 正确分析了所有套件中的所有测试用例，没有 FP 或 FN。也就是说，每个 TP 至少被 1 个模块检测到，而 TN 没有被检测到。我们手动验证 TP 案例的根本原因报告是否正确识别了有缺陷的功能，并且建议可以防止内存损坏

- ARCUS 在 RIPE 案例上非常准确，因为有多种机会检测溢出。例如，损坏返回指针的整数溢出可以在寄存器回绕时由整数溢出模块检测，或者在指针被覆盖时由堆栈溢出模块检测

- 对于 Juliet 案例测试的模块，它们的能力不重叠并且产生相同的数字和整个表格一样。对于与 RIPE 相关的策略，我们发现符号 IP 检测的平均准确率为 92.9%，而整数溢出检测的准确率为 69.5%

- Can ARCUS locate and analyse real-world exploits?

CVE / EDB	Type	Program	# BBs	Size (MB)	ΔRoot Cause	ΔAlert	Located	Has Patch	Match
CVE-2004-0597	Heap	GIMP (libpng)	41,625,163	56.0	247	1	Yes	[61]	Yes†
CVE-2004-1279	Heap	jpegtoavi	67,772	0.65	26,216	1	Yes	No	-
CVE-2004-1288	Heap	o3read	74,723	0.65	33,211	1	Yes	[62]	Yes
CVE-2009-2629	Heap	nginx	300,071	1.10	28	33,824	Yes	[63]	Yes
CVE-2009-3896	Heap	nginx	283,157	1.10	59	16,821	Yes	[64]	Yes
CVE-2017-9167	Heap	autotrace	75,404	1.01	1,828	2	Yes	No	-
CVE-2018-12326	Heap	Redis	291,275	1.20	8	234	Yes	[65]	Yes
EDB-15705	Heap	ftp	260,986	0.85	19,322	2	Yes	No	-
CVE-2004-1257	Stack	abc2mtex	53,490	0.67	6,319	1	Yes	No	-
CVE-2009-5018	Stack	gif2png	90,738	1.09	1,848	1	Yes	[66]	Yes
CVE-2017-7938	Stack	dmitry	100,186	0.71	4,051	14,402	Yes	No	-
CVE-2018-12327	Stack	ntpq	374,830	1.85	122,740	77,990	Yes	[67]	Yes
CVE-2018-18957	Stack	GOOSE (libiec61850)	65,198	0.71	94	30	Yes	[68]	Yes
CVE-2019-14267	Stack	pdfresurrect	128,427	0.66	83,123	1	Yes	[69]	Yes
* EDB-47254	Stack	abc2mtex	53,490	0.67	6,566	-	Yes	No	-
EDB-46807	Stack	MiniFtp	60,849	0.69	335	107	Yes	No	-
CVE-2006-2025	Integer	GIMP (libtiff)	78,419,067	55.0	3	8	Yes	[70]	Yes
CVE-2007-2645	Integer	exif (libexif)	67,697	0.97	1	7	Yes	[71]	Yes
CVE-2013-2028	Integer	nginx	809,977	2.00	1	25,268	Yes	[72]	Yes
CVE-2017-7529	Integer	nginx	1,049,494	1.10	2	780,404	Yes	[73]	Yes
CVE-2017-9186	Integer	autotrace	75,142	1.00	1	1	Yes	No	-
CVE-2017-9196	Integer	autotrace	74,695	1.03	1	203	Yes	No	-
* CVE-2019-19004	Integer	autotrace	132,302	1.02	1	-	Yes	No	-
CVE-2017-11403	UAF	GraphicsMagick	2,316,152	4.61	38	1	Yes	[74]	Yes
CVE-2017-14103	UAF	GraphicsMagick	2,316,133	4.61	38	1	Yes	[74]	Yes
CVE-2017-9182	UAF	autotrace	132,302	1.02	296	58,058	Yes	No	-
* CVE-2019-17582	UAF	PHP (libzip)	5,980,255	6.40	49	-	Yes	[75]	Yes
CVE-2017-12858	DF	PHP (libzip)	5,980,255	6.40	51	719	Yes	[75]	Yes
* CVE-2019-19005	DF	autotrace	132,302	1.02	57,859	-	Yes	No	-
CVE-2005-0105	FS	typespeed	127,209	0.74	1	1	Yes	[76]	Yes
CVE-2012-0809	FS	sudo	108,442	0.69	1	1	Yes	[77]	Yes
		Average:	**4,568,619**	**5.07**	**11,722**	**36,804**			

* New vulnerability discovered by ARCUS.

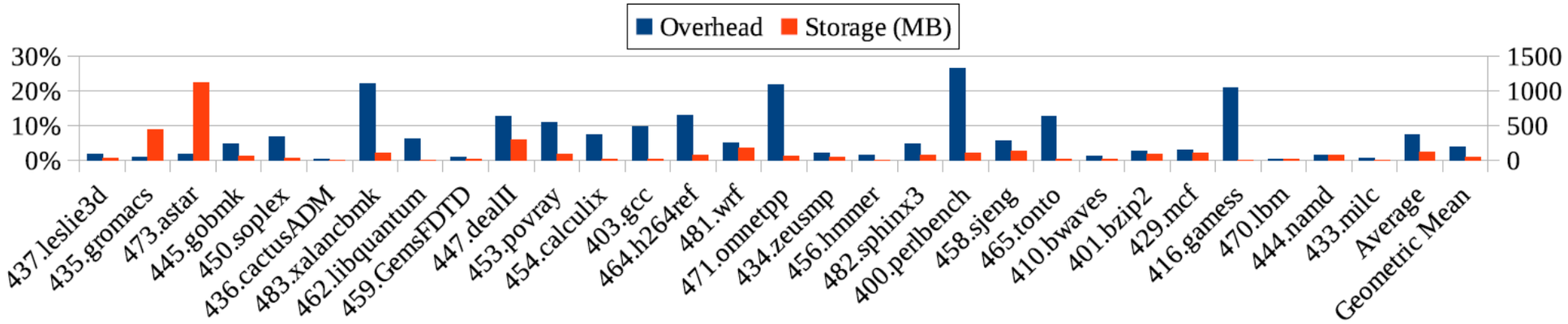
† Equivalent to applied patch.

- Are ARCUS' root cause reports consistent with real world advisories and patches?

- 通过手动将它们与公共漏洞进行比较评估漏洞利用报告的质量

- 在这些案例中，我们使用补丁来进一步验证 ARCUS 的质量通过手动确认他们识别出相同的代码

- Is ARCUS feasible to deploy in terms of runtime and storage overhead?



跟踪 SPEC CPU 基准的性能开销和存储大小。 平均开销为 7.21%，几何平均值为 3.81%。 平均跟踪大小为 110 MB，几何平均值为 38.2 MB。

Results

- 我们证明我们的方法可以构建符号程序状态并分析几类严重和普遍的软件漏洞

- 我们对 27 个漏洞和 9,000 多个 Juliet 和 RIPE 测试用例的评估表明，ARCUS 可以自动识别所有测试漏洞的根本原因，在此过程中发现 4 个新漏洞

- ARCUS 在 SPEC 2006 CPU 基准测试中产生了 7.21% 的性能开销，并可扩展到从超过 810,000 行 C/C++ 代码编译的大型程序

Thanks

Code: https://github.com/carter-yagemann/ARCUS